A Parallel SPH Implementation on Shared Memory Systems

Yaidel Reyes López
Dept. Computer Science, KU Leuven, Leuven, Belgium
CIMCNI, UCLV, Santa Clara, Cuba

yaidel.reyeslopez@cs.kuleuven.be

Dirk Roose
Dept. Computer Science
KU Leuven
Leuven, Belgium
dirk.roose@cs.kuleuven.be

***Abstract*—We present a parallel implementation of SPH for shared memory computers. Our approach is based on domain decomposition and space filling curves (SFC). The particles are sorted and assigned to threads according to the Z-curve. This ensures per thread local storage of most frequently accessed data, avoids NUMA-unfriendly memory allocations, reduces data races and allows efficient calculation of symmetric inter-particle forces. We describe a simple and inexpensive dynamic load balancing algorithm. Finally, we present strong and weak scalability results of the implementation, and we identify sources of overhead.**

I. Introduction

The Smoothed Particle Hydrodynamics (SPH) method is currently used in many application areas including fluid dynamics, solid mechanics, astrophysics, coastal and marine engineering, and others. Compared to other simulation techniques, SPH simulations are computationally expensive: a large number of particles is required to achieve good accuracy; small time steps are needed, a limitation that results form utilizing explicit time integrators; interacting particles must be identified in every time step due to the particle movement. For that reason much attention has been paid to the parallelization of the method, mainly considering highly parallel systems, i.e. computer clusters and Graphic Processing Units (GPU). Examples are the open-source code DualSPHysics [1] (http://www.dual.sphysics.org) and the code SPH-flow [2] (http://www.sph-flow.com).

Although some work has been done on the parallelization of SPH on shared memory systems [3], [4], the attention to this type of architecture has been limited. However the number of cores per processors quickly increases, and will continue to increase. This, together with multi-socket boards provided with fast interconnects featuring cache-coherent non-uniform memory access (ccNUMA), provides shared memory system with large calculation power.

Despite the fact that a shared memory space suggests easier parallelization, there are several factors that are a detriment to efficiency. In particular, sharing of cache memory may lead to extra cache misses, and on systems with NUMA, improper data placement in memory can lead to significant latencies if threads often access data that reside far from them, possibly also causing bandwidth bottlenecks. Also, different synchronization mechanisms may be required in order to avoid data races, which in turn introduces parallel overhead.

In the next section we briefly discuss modern shared memory architectures and describe a strategy to efficiently use caches and avoid NUMA-unfriendly memory allocations. Then we elaborate on the domain decomposition technique and the approach to handle data accesses at the boundary of the subdomains. Finally, we present and discuss the parallel algorithm. Section III presents a scalability study for a breaking dam problem, and we analyze the load balance during the different phases of the simulation loop.

II. Parallelization on Shared Memory

In order to reduce the computational cost, SPH implementations exploit the compact condition that must be satisfied by the kernel, i.e. every particle interacts only with particles inside a local neighborhood with radius κh, where κ is a constant that depends on the kernel function, and h is the smoothing distance. Different data structures can be used to efficiently identify interacting particles.

For simulations that discretize the physical domain using particles with (approximately) the same smoothing distance, the most efficient data structure is an underlying regular grid. Every cell in the grid has information regarding the particles that lay in it. The cell size is chosen so that interacting particles are only in direct neighboring cells. Then, the SPH simulation loop generally consist of the following steps:

1) assign particles to grid cells depending on their current position,
2) determine interacting particles using the information on the grid,
3) compute flux terms, i.e. process the interactions,
4) perform time integration.

Flux terms can be computed in two ways: (a) for every particle i find the list of neighbors and compute the contribution of i's neighbors to i; (b) find all interacting pairs of particles, then for every pair (i, j) account for the reciprocal contributions of particles i and j , exploiting the symmetry of the interactions.

The parallelization of the first approach is simple. It is an embarrassingly parallel algorithm given that every particle can

be processed independently without race conditions. However, it does not allow the reciprocal calculation of interactions, thus requires nearly twice the amount of calculations to compute the flux terms if compared to the second approach. The drawback of the second approach is the challenges it poses for shared memory parallelization. Since one particle is included in several interaction pairs, parallelization of the computation of flux terms is not straightforward due to data races.

In [5] a slowdown of 1.6 was reported after replacing the approach (b) by (a) to implement a CPU-GPU hybridization. This severely limits the parallel speedup that can be achieved. For that reason, throughout this paper we will focus on the computationally cheaper second approach.

A. General considerations

Current parallel shared memory architectures have multiple cores, each of which has access to a hierarchy of memories $(m_0, m_1, ..., m_n)$, with m_i being in a sense subordinated to the next higher level m_{i+1}. Often multiple caches are placed at the lowest levels in between the cores and the main memory[1]. The lower the level, the faster the prefetching times, but also the smaller the capacity. Every core may have individual caches, and groups of cores share higher level caches or a single memory controller. On hierarchical memory systems it is important to take advantage of spatial and temporal data locality to allow an efficient use of caches by reducing prefetchings from higher levels on the memory hierarchy.

Systems with multi-socket boards feature ccNUMA shared memory architectures. Every socket holds a multi-core processor that has its own local memory module, i.e. a NUMA node. Every core within the NUMA node can directly access the local memory. At the same time, every core can access any of the memory modules of other NUMA nodes, but the access is slower.

These elements must be considered to take optimal advantage of current shared memory systems. However, modern programing languages assume only two levels of memory, i.e. main memory and disk storage, leaving to the hardware the responsibility of moving data between memory and caches. Thus, the programmer needs to find optimal processing patterns or allocation strategies that implicitly result in efficient cache use and reduce the transfer of data between different levels of the memory hierarchy.

B. Data Locality

Space filling curves (SFC) [6] offer a means to sequentialize multidimensional data while still preserving spatial locality properties. SFC have been used to increase cache efficiency in many numerical applications, including SPH simulations [3]. In this section we explain the application of SFC to preserve data locality in our implementation. In particular, we use the Z-curve [7] because the indices, i.e. the Morton codes or Z-indices, can be computed efficiently from multidimensional coordinates using bit interleaving.

[1]Actually, the registers are at a lower level than caches, offering the fastest possible access. But this does not affect the considerations presented here.

The grid cells are stored in memory following the Z-curve, i.e. every grid coordinate maps to a Z-index and cells are placed in memory in increasing order of the Z-indices. This ensures that neighboring cells are very likely to be close in memory.

However, in SPH most of the calculations are performed on interacting particles. Therefore, interacting particles must be kept close in memory. This can be achieved by sorting the array of particles according to the Z-index of the cell on which each particle lies. The Lagrangian nature of SPH implies that particles have to be sorted regularly to keep locality throughout the entire simulation.

To further exploit having the array of particles sorted, a grid cell with Z-index c_i only stores the array index of the first particle with Z-index equal to c_i. This is sufficient to determine all the particles in the cell. As a consequence, particles have to be sorted every time before updating the grid. Although sorting the particles seems an excessive amount of work, the following considerations make clear that this is not necessarily the case.

In order to be stable, SPH time integration must follow the Courant-Friedrichs-Lewy (CFL) condition which states that a particle can't travel farther than half the smoothing distance in one time step. Hence, a sorted array of particles becomes only slightly unsorted from one time step to the other. Moreover, several particles can be in the same cell, having the same sorting key. A sorting algorithm well suited for this situation must be chosen. In [3] the authors show how for this problem insertion sort outperform radix sort. The main two reasons are: (a) insertion sort is stable, i.e. particles with the same key are not swapped; and (b) it is adaptive, for nearly sorted arrays it has a complexity of $O(N + d)$, where d is the number of swaps, which is small due to the CFL condition.

Since particles are stored in a relatively large data structure with several fields, resorting the array of particles every time is not optimal. Instead, in [3] the authors propose to keep an array of handles, where a handle contains a pointer to a particle and its corresponding Z-index. Every time the grid has to be updated, the array of handles is sorted. Due to the minimal memory usage of this structure, sorting the handles is much faster than sorting the particles. However, in order to keep spatial locality the array of particles must still be sorted. Since particles move relatively slow, this can be done after having sorted the handles n_{sort} times, where n_{sort} is a problem dependent number.

Note also that an approach similar to the Verlet list method [8] can be used so that the information required to find interactions is valid for a few time steps.

C. NUMA-friendly memory allocation

On shared memory systems with NUMA, another element to take into account is data placement in memory. If non-local memory is accessed frequently, cores might spend much time idling, waiting for the memory access to be completed.

Data placement in memory can be performed explicitly or implicitly. The former requires the utilization of APIs, e.g. the

libnuma library (http://oss.sgi.com/projects/libnuma/), placing the burden of handling data placement on the programmer. The latter transparently handles data placement using one of the following policies: *local to first request*, where the memory is local to the thread that request the allocation; *local to first access*, wait for the first memory access before committing on memory assignment, in this way the memory can be made local to the first thread that access it (on most systems the default policy); or *interleave*, which result in the data evenly distributed across all nodes. Either policy can be advantageous depending on the application, and can be set system-wide or specifically for a given application using tools like **numactl**, provided with **libnuma**.

In our work, in order to ensure that the data most frequently accessed by a thread is local to it, we use a domain decomposition strategy. The physical domain is decomposed into subdomains and every thread executes the computations corresponding to the particles in a subdomain. The threads allocate and initialize the memory necessary for its subdomain. Using implicit data placement with the *local to first access* policy, we ensure that this memory is local to the thread.

One implicit advantage of decomposing the domain is that race conditions are largely avoided: all thread-local data can be safely processed in parallel. However, as with implementations on distributed memory systems, efficient dynamic load balance and treatment of subdomain boundaries is very important.

D. Domain decomposition

In SPH, the number of particles is a good indicator of the amount of work to be performed. Therefore, we decompose the domain so that every thread's subdomain has approximately the same number of particles, and we exploit the fact that both the particles and the underlying grid are sequentialized according to the same Z-curve.

The decomposition of the 2D or 3D domain is achieved by dividing the array of N sorted particles in P chunks of about N/P particles, where P is the number of threads. Due to the locality properties of the Z-curve, the P chunks of particles correspond to compact partitions [6]. Figures 1 and 2 schematically represent this procedure for a 2D problem. The particles, i.e. colored circles, are discretizing the volume of fluid represented by the light blue region, while the grid covers the whole square simulation domain. The division in $P = 4$ chunks results in the colored subdomains. Since the grid cells are stored according to the same Z-curve, the partitions of the grid are directly determined from the partitions of the particles.

Particles lying on one grid cell should not be assigned to two different threads since this would result in too many special cases while implementing the neighboring query. Therefore, we require that all the particles with the same Z-index belong to the same partition. Hence, the partition corresponding to thread p has $\lceil N/P \rceil \pm k_p$ particles. Notice that the term k_p introduces some load imbalance that can be neglected because k_p is smaller than the maximum number of particles per cell, which is very small compared to $\lceil N/P \rceil$.

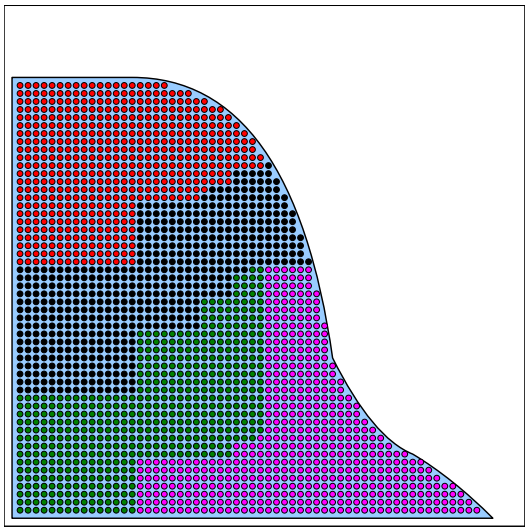

Fig. 1. Schematic 2D domain decomposition for $P = 4$. Particles (circles) are sorted according to the Z-curve. Each subdomain is shown in a different color.

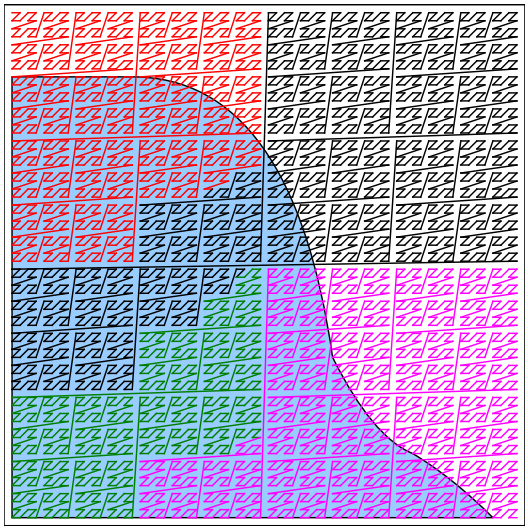

Fig. 2. Schematic 2D domain decomposition for $P = 4$. Every turning point of the Z-curve corresponds to a grid cell. Each subdomain is shown in a different color.

During the parallel processing, every thread requires information from neighboring subdomains, thus having to access non-local data. The amount of non-local data that must be accessed is proportional to the number of particles at the boundary, and the amount of local data that must be processed is proportional to the number of particles in a partition. Therefore, keeping the boundary/volume ratio of the partitions as small as possible is ideal in order to minimize dependencies on non-local particles. Although the locality properties of the Z-curve, and SFC in general, ensure that the domain decomposition technique described in this section results in compact partitions, there is no control over the boundary of the partitions. It can be seen in figures 1 and 2 that the partitions are not optimal in this respect. This is a drawback of this

simple and computationally efficient domain decomposition technique.

E. Handling non-local data

So far, we have defined how to partition the particles according to the Z-curve. However, in order to compute the flux terms of particles near the boundary of one partition, information from particles in neighboring partitions is required. We now discuss some possible approaches and elaborate on the strategy that we follow in this work.

The following approaches could be applied to exchange data between partitions:

(a) A procedure similar to that used on distributed memory systems can be applied, namely using a local buffer to copy the non-local data that has to be accessed. This implies that data has to be duplicated. Moreover, copied particles are treated as a kind of ghost particles, i.e. every thread computes the flux terms only for the local particles, not for the copied ghost particles. Hence the symmetry of the interaction terms can't be exploited. Non-local data is accessed only once, i.e. for copying it to the local memory.

(b) We can further exploit the shared memory capabilities, i.e. there is no need for copying non-local data because it is still accessible. As in the previous approach, non-local particles are considered as read-only ghost particles, i.e. a thread only modifies local data. This avoids the data duplication, but implies several accesses to non-local data.

These first two approaches implicitly avoid data races by not exploiting the symmetry of the interactions at the boundary of the partitions. If we want to take advantage of the symmetry of the flux terms, then synchronization is required when processing particles at the boundaries. Additionally this also implies that non-local data is accessed several times for both reading and writing. The following two procedures use this strategy:

(c) Exclusive access to the particles near the subdomain boundaries can be granted by using locks. This might result in data contention if several threads try to access the same particle at the same time. Nevertheless, locks are only necessary for particles near the boundary, thus the overhead might be acceptable.

(d) In [4] the authors present a coloring procedure to avoid data races when computing symmetric flux terms. This algorithm can be applied to process the interactions of particles near the boundaries of the subdomains. The particle interactions are divided in groups that are processed in different phases to avoid data races. After each phase, a global synchronization is required.

Clearly, each of these approaches have advantages and disadvantages. In this work we focus on the approach (d), and leave the study of the other three, and a comparison between them for future research.

We describe and depict the application of the coloring procedure presented in [4] in 2D. The extension to 3D is intuitive and can be found in [4]. The color of the cell (i, j) is determined by the pair $(i \bmod 2, j \bmod 2)$, which results in four possible colors. Consider the interaction pattern shown in figure 3 (bottom-right). The arrows represent the inter-cell particle interactions, and the dot the intra-cell particle interactions. The application of such pattern on cells of the same color results in disjoint sets of particle interactions that can be safely processed in parallel. Figure 3 shows the application of the pattern on the colored cells. In the figure the entire grid is colored to illustrate the procedure, but the coloring is only applied on boundary cells.

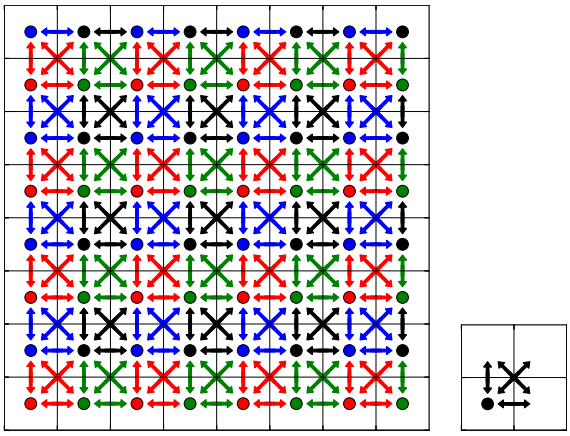

Fig. 3. Coloring procedure. Fully colored grid (left) and interaction pattern (right). The color of the cell is given by the color of the dot.

F. Final algorithm

We now put together all the elements previously mentioned. For the most important parts of the final algorithm, implementation details are provided in form of pseudo code.

1) Precomputing particle interactions: In SPH, many models require to process the particle interactions more than once per time step. This is certainly the case for any SPH implementation that computes a corrected kernel or a gradient correction [9], [10]. Since computing all the particle interactions is an expensive task, computing them on the fly every time the interactions have to be processed is not efficient. Instead, all possible interactions are precomputed and kept in interaction lists. We also follow an approach similar to the Verlet list, i.e. a slightly larger support domain is considered so that the interaction lists are valid for several time steps.

The procedure to precompute the particle interactions is shown in Algorithm 1, which is executed concurrently by every thread. The procedure *PatternInCell(c)* returns all pairs of cells that are obtained by applying the interaction pattern on cell c, including the pair (c, c) (see figure 3). Then, every pair of cells (c_0, c_1) is processed; if both cells are local to the thread then all the interactions between particles in c_0 and particles in c_1 are added to the list of local interactions $iListLocals$ by the procedure *AddInteractionsToList*; otherwise, the interactions are added to the list $iListCol[col]$ depending on the color col of the cell.

Algorithm 1 Precompute interaction lists

```
for each c in localCells do
    for each (c_0, c_1) in PatternInCell(c) do
        if IsLocal(c_0) and IsLocal(c_1) then
            AddInteractionsToList(iListLocal, c_0, c_1)
        else
            col = GetColor(c)
            AddInteractionsToList(iListCol[col], c_0, c_1)
```

Every partition of the grid is determined by a half open interval $[c_b, c_e)$, where c_b is the Z-index of the first particle in the partition, and c_e is the Z-index of the first particle in the next partition. Thus, given a cell's Z-index it is easy to determine whether it is local to the thread. Moreover, if the cell is not local, the partition on which the cell is located can be easily and efficiently found by performing a binary search on the P intervals.

2) Processing Particle Interactions: In accordance with Algorithm 1, local interactions involve two particles belonging to the same subdomain, and are processed by the thread assigned to that subdomain. Non-local interactions involve two particles belonging to different subdomains, and are processed by the thread assigned to the cell on which the interaction pattern was applied.

The procedure executed concurrently by every thread to process particle interactions is shown in Algorithm 2. First, all local interactions are processed, afterwards the interactions at the subdomain boundaries are processed in different phases according to their color. The global synchronizations are required to avoid race conditions. For a d-dimensional problem, 2^d synchronizations are necessary.

Algorithm 2 Processing particle interactions

```
for each inter in iListLocal do
    ProcessInteraction(inter)
barrier                          ▷ global synchronization
for each col in colors do
    for each inter in iListCol[col] do
        ProcessInteraction(inter)
    barrier                      ▷ global synchronization
```

3) Final algorithm: In figure 4 the general scheme of the simulation is presented. We describe now this scheme, focusing mainly on data distribution and processing of the particles by the different threads. The arrays represented in the figure correspond to the particle arrays. Since the partitions of the grid are directly determined from the partitions of the particles, it is not necessary to include the grids in the figure.

The initial unsorted array of N particles is distributed, assigning approximately N/P particles per thread. At this point, the main loop of the simulation starts, and it is concurrently repeated by each thread until the stop condition is reached. Since particles are initially unsorted, the initial distribution does not result in compact partitions, but this is immediately solved after the first global sorting of the particles.

The global sorting of the particles is executed in two phases: (a) every thread sorts the local particles, and (b) a parallel sorting algorithm is performed. For the local sorting every thread uses insertion sort except for the first time because the particles are totally unsorted, therefore we use introsort, which is an hybrid sorting algorithm that combines heapsort and quicksort and has a better average and worst case complexity. To complete the sorting, the locally sorted arrays are combined and sorted globally by parallel odd-even transposition sort.

Once the particles are globally sorted, we need to avoid that particles lying on the same grid cell belong to different partitions. This procedure is represented in figure 4 by the colored circles. After the sorting, partition 0 and partition 1 have particles that belong to the same grid cell (represented with the green circles). This situation is solved by applying a cell fitting procedure, which requires the movement of some particles from one partition to another, resulting in adding or removing k_p particles for every partition p. Since k_p is smaller than the number of particles per grid cell, $k_p \ll N/P$ and the imbalance introduced by the cell fitting procedure is negligible. However, since this cell fitting procedure is executed in every time step, the imbalance may increases with increasing number of time steps. Therefore, a load rebalance is necessary after a number of iterations of the simulation loop.

The dynamic rebalancing procedure is simple and not costly. Every partition exchanges a number of particles with its predecessor and successor in order to level out the number of particles per thread. Inserting at, or removing from, the end of an array has a constant complexity. Performing the same operations at the beginning of the array requires shifting all the element of the array, which has a cost of $O(N/P)$. However we avoid this by applying the following strategy. Space for k_{max} particles is left available at the beginning of the local arrays, where k_{max} is an estimate of the maximum number of particles per cell. Also, if some particles have to be moved from partition p to partition $p-1$, the other particles on partition p are not shifted to the left, instead we keep the space of the moved particles available for future insertions at the beginning. This procedure results in a constant insertion and removal time from the beginning of the local arrays.

Finally, the procedures required to perform a *time step* in SPH are executed: (a) if the current interaction lists are no longer valid, then recompute the interaction lists, (b) process interactions to compute flux terms according to the model, and (c) perform time integration and move to the next time step. The latter phase only requires local data access and the execution time is proportional to the number of particles.

III. Results

In this section we present the results obtained with the presented parallel implementation for the simulation of a free surface flow, i.e. a breaking dam simulation. We follow the Weakly Compressible SPH approach [11] with the XSPH

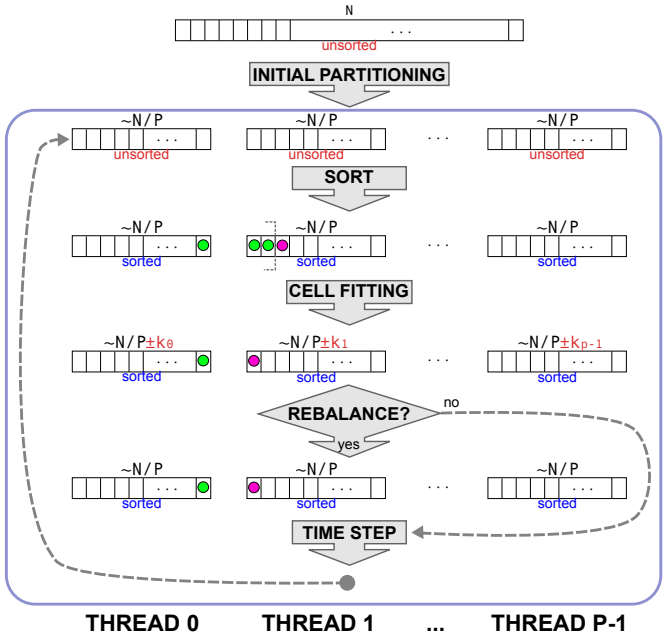

Fig. 4. Scheme of the simulation. The rounded box corresponds to the simulation loop. The scheme shows the processing and distribution of particles for every thread.

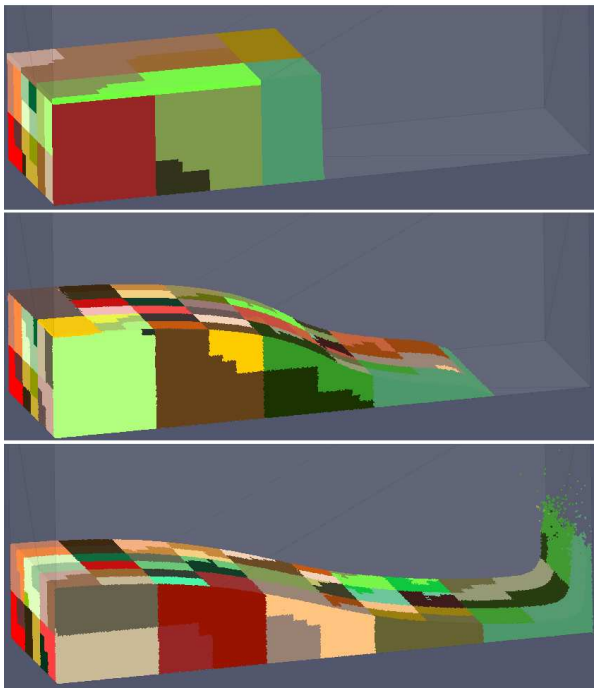

Fig. 5. Frames corresponding to time steps 1600, 16000 and 25600 of a breaking dam simulation with 1920000 fluid particles on 64 cores with 64 threads. Color represents the different partitions.

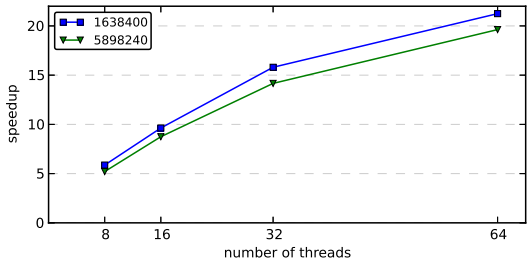

Fig. 6. Strong scalability study for 1638400 and 5898240 particles.

correction to the velocity field. All the experiments were run in an 8 sockets NUMA machine, where every socket contains an 8 core Intel Xeon E7-2830 Westmere-EX at 2.13GHz, totaling 64 cores. For scalability tests, we compare with the sequential algorithm that exploits the symmetry of the flux terms.

Figure 5 shows some frames of the breaking dam simulation. This simulation was performed using 1920000 fluid particles. The colors represent the different partitions corresponding to the 64 threads used. It can be seen, especially in the top image, that some partitions have a boundary/volume ratio that is far from optimal. This leads to a larger than optimal fraction of the non-local interactions. This is partially due to the fact that the Z-curve traverses the grid that covers the whole simulation domain, i.e. the space in which the particles may reside in any of the time steps.

A. Strong scalability

We performed a strong scalability study, keeping the number of particles constant while increasing the number of cores from 8 to 64. A breaking dam simulation with particles initially arranged to form a block of water of $0.8m \times 1m \times 1m$ is run for 8000 iterations. The results are shown in figure 6 for 1638400 and 5898240 particles wit time steps $\nabla t = 1.18 \cdot 10^{-5}$ and $\nabla t = 1.76 \cdot 10^{-5}$ respectively. The speedups are very similar for both simulations, however the simulation with more particles resulted in slightly smaller speedups.

B. Weak scalability

For the weak scalability study, we keep the number of particles per thread constant and run the breaking dam simulation for 8000 iterations. In figure 7 we present the execution time and the parallel efficiency obtained for different numbers of particles per thread. We observe a decrease in parallel efficiency when the number of threads increases. Increasing the number of cores beyond 8 leads to NUMA-effects since some of the non-local data accesses occur at another socket. At present, we do not have a full understanding of the jumps in execution time and efficiency.

Below, we present a detailed analysis of the work load balance of the different phases of the simulation loop.

C. Analysis of the load balance

The distribution of the work over the threads is based on domain decomposition ensuring that each subdomain contains approximately the same number of particles (see section II-D). However, a large fraction of the execution time of a time step is spent in processing the particle interactions. As a result, even if the particles are nearly equally distributed over the threads, the distribution of the particle interactions over the threads may be

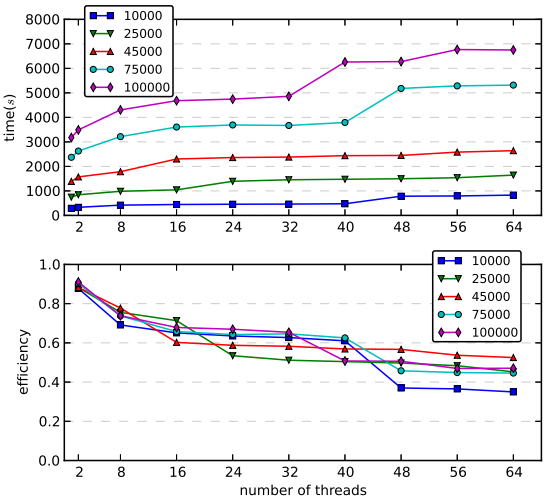

Fig. 7. Weak scalability study. Execution time (top) and parallel efficiency (bottom).

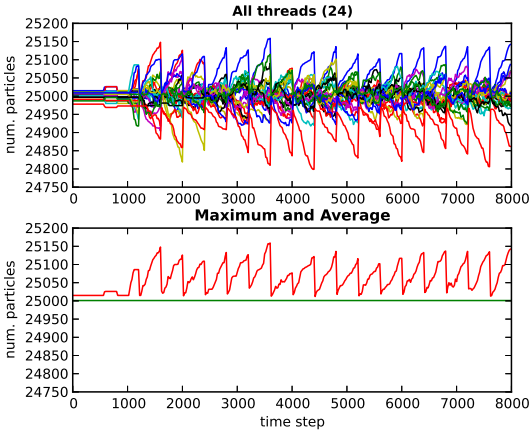

Fig. 8. Number of particles per thread.

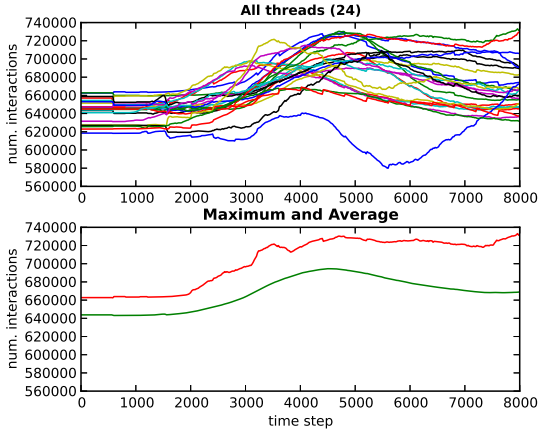

Fig. 9. Number of local interactions per thread.

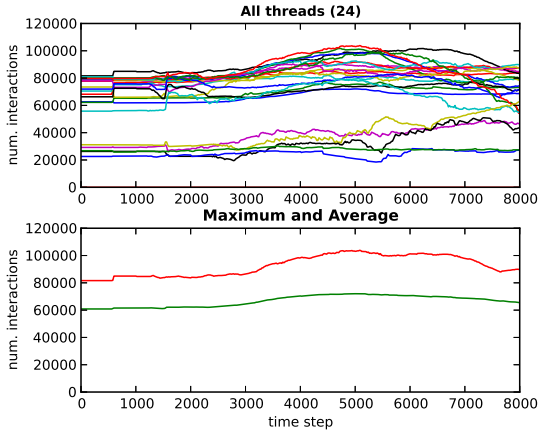

Fig. 10. Total number of non-local interactions (the sum of all non-local interactions per color) per thread.

more unbalanced. In addition, when processing the non-local particle interactions, these interactions are divided in groups (cf. coloring scheme) and each group is processed separately, with a global synchronization in between. Hence, the load balance of processing the non-local interactions depends on the balance of the number of non-local interactions per color.

To assess the overall work load balance, we have performed detailed measurements during the breaking dam simulation, up to time step 8000. We show the results for $P = 24$ and 25000 particles per thread.

Figure 8 shows that the number of particles per thread remains nearly constant during the simulation. The difference between the average and the maximum number of particles per thread slowly grows in between rebalancing steps, but remains smaller than 0.6%, indicating that the rebalancing works well.

Figure 9 shows the number of elements in the local interaction list, which is updated every 20 time steps. This number increases due to the particles moving slightly closer to each other after leaving the initial steady state. The difference between the average and the maximum number of local interactions varies from $\sim 3\%$ during the first time steps to $\sim 6\%$ in later time steps. So the load imbalance during the local interaction processing phase remains limited.

Figure 10 shows the total number of elements in the non-local interaction lists, i.e. the sum of the sizes of the lists in *iListCol* (see Algorithm 1). Comparing the maximum with the average over the threads indicates that the load imbalance for processing these interactions grows to more than 20%, much larger than for processing local interactions. Additionally, a more detailed study shows that the imbalance per color is higher, from $\sim 25\%$ to $\sim 35\%$. However, the total number of non-local interactions is only about 10% of the local interactions, limiting the effect of these imbalances.

In order to estimate the overhead associated to the synchronizations, we measure the execution time of processing the different interaction lists. In Table I we show timings concerning the processing of the interactions for computing the continuity density equation, the Euler momentum equation and the artificial viscosity. We show the times without synchronization, i.e. only processing time, and the times

TABLE I
MAXIMUM AND AVERAGE TIMES FOR PROCESSING INTERACTIONS

	without synchronization			with synchronization		
	local	non-local	%	local	non-local	%
Max.	0.08079	0.01057	13.1%	0.09362	0.01265	13.5%
Avg.	0.07679	0.00761	9.9%	0.08623	0.01200	13.9%

Max. and Avg. refer to the per thread values.
% indicates non-local values as percentage of the local.

including synchronization, which accounts for time idling due to imbalances and the actual cost of the global barriers. For non-local interactions we consider the sum of the execution times required for each colored list. The numbers in the table are the execution times per time step, averaged from time step 500 to 8000. We ignored the first time steps since the execution times in these time steps vary much and are nearly an order of magnitude higher than in later time steps.

Neglecting synchronization, the average time required to process all non-local interactions represent 9.9% of the time required to process local interactions. This is in good agreement with the numbers of local and non-local interactions shown in figures 9 and 10, and shows that the NUMA effects while processing the non-local interactions does not have a significant impact on the execution times. This may be due to the compact distribution of the threads over the NUMA nodes[2], which means that neighboring subdomains are likely to be assigned to threads in the same node. This percentage however increases up to about 14% when including the synchronization time due to the global barrier and the load imbalance per color.

IV. CONCLUSION

We described an algorithm for parallelizing SPH on shared memory systems. The application of domain decomposition combined with the Z-curve offers several advantages: the partitioning algorithm is simple, and dynamic load balancing is inexpensive and effective in terms of the number of particles per subdomain; NUMA-friendly memory allocation is achieved by performing per thread allocation of memory to store the data corresponding to the thread's partition; spatial data locality is provided, which in turn results in efficient cache memory use.

However, we identified also a few drawbacks. Storing the grid according to the Z-curve requires that the dimensions of the grid are a power of two, which for some problems may result in a very large grid. Hence, the Z-curve can not by adjusted to cover only the particles. Therefore, although the partitioning results in fairly compact subdomains, for some geometries the boundary/volume ratio may be far from optimal.

[2]e.g. threads 0,1,...,7 are given to the NUMA node 0, thus they share the same local memory. Accordingly, threads 8,9,...,15 are given to NUMA node 1, and so on.

ACKNOWLEDGMENT

The authors would like to thank the Flanders Exascience Lab (Intel Labs Europe) for access to the 64 cores Intel shared memory system.

REFERENCES

[1] D. Valdez-Balderas, J. M. Domínguez, B. D. Rogers, and A. J. Crespo, "Towards accelerating smoothed particle hydrodynamics simulations for free-surface flows on multi-GPU clusters," *Journal of Parallel and Distributed Computing*, vol. 73, no. 11, pp. 1483 – 1493, 2013, novel architectures for high-performance computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731512001712

[2] D. Guibert, M. De Leffe, G. Oger, and J. Piccinali, "Efficient parallelization of 3D SPH schemes," in *7th SPHERIC workshop*, 2012, pp. 259–265.

[3] M. Ihmsen, N. Akinci, M. Becker, and M. Teschner, "A parallel SPH implementation on multi-core CPUs," *Computer Graphics Forum*, vol. 30, no. 1, pp. 99–112, 2011. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2010.01832.x

[4] J. Willkomm and H. Bücker, "Parallel summation of symmetric inter-particle forces in smoothed particle hydrodynamics," in *Meshfree Methods for Partial Differential Equations V*, ser. Lecture Notes in Computational Science and Engineering, M. Griebel and M. A. Schweitzer, Eds. Springer Berlin Heidelberg, 2011, vol. 79, pp. 235–248. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16229-9_15

[5] G. Oger, E. Jacquin, M. Doring, P. Guilcher, R. Dolbeau, P. Cabelguen, L. Bertaux, D. Le Touzé, and B. Alessandrini, "Hybrid CPU-GPU acceleration of the 3-D parallel code SPH-flow," in *5th International SPHERIC Workshop*, 2010.

[6] M. Bader, *Space-Filling Curves - An Introduction with Applications in Scientific Computing*, ser. Texts in Computational Science and Engineering. Springer-Verlag, 2013, vol. 9. [Online]. Available: http://link.springer.com/book/10.1007/978-3-642-31046-1/

[7] G. Morton, "A computer oriented geodetic data base; and a new technique in file sequencing," IBM Ltd., Ottawa, Ontario, Canada, Tech. Rep., 1966.

[8] L. Verlet, "Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules," *Physical Review*, vol. 159, no. 1, pp. 98–103, Jul. 1967. [Online]. Available: http://dx.doi.org/10.1103/physrev.159.98

[9] J. Bonet and T. S. L. Lok, "Variational and momentum preservation aspects of smooth particle hydrodynamic formulations," *Computer Methods in applied mechanics and engineering*, vol. 180, no. 1-2, pp. 97–115, 1999. [Online]. Available: http://dx.doi.org/10.1016/S0045-7825(99)00051-1

[10] J. K. Chen and J. E. Beraun, "A generalized smoothed particle hydrodynamics method for nonlinear dynamic problems," *Computer Methods in Applied Mechanics and Engineering*, vol. 190, pp. 225–239, 2000.

[11] J. Monaghan, "Simulating free surface flows with SPH," *Journal of Computational Physics*, vol. 110, no. 2, pp. 399–406, February 1994. [Online]. Available: http://dx.doi.org/10.1006/jcph.1994.1034